Lecture 16:
Regex, FSA, Morphology, FST

Ling 1330/2330 Intro to Computational Linguistics
Na-Rae Han, 10/24/2023

Outline

- Regular expressions wrap
- FSA
- Morphology and FST
  - Jurafsky & Martin (2nd Ed!) Ch.3 Words and Transducers
  - Hulden (2011) Morphological analysis with FST

    ← foma!

Tokenization through re.split(), re.findall()

```
>>> sent = "It's 5 o'clock somewhere. Why don't we drink a martini."
>>> sent.split()
    ["It's", '5', "o'clock", 'somewhere.', 'Why', "don't", 'we', 'drink',
    'a', 'martini.']
>>> re.split(r'\s+', sent)
    ["It's", '5', "o'clock", 'somewhere.', 'Why', "don't", 'we', 'drink',
    'a', 'martini.']
>>> re.split(r'[ eo]', sent)
    ["It's", '5', '', "'cl", 'ck', 's', 'm', 'wh', 'r', '.', 'Why', 'd',
    "n't", 'w', '', 'drink', 'a', 'martini.']
>>> re.split(r'\W', sent)
    ['It', 's', '5', 'o', 'clock', 'somewhere', '', 'Why', 'don', 't',
    'we', 'drink', 'a', 'martini', '']
>>> re.split(r'\W+', sent)
    ['It', 's', '5', 'o', 'clock', 'somewhere', 'Why', 'don', 't', 'we',
    'drink', 'a', 'martini', '']
>>> re.findall(r'\w+', sent)
    ['It', 's', '5', 'o', 'clock', 'somewhere', 'Why', 'don', 't', 'we',
    'drink', 'a', 'martini']
```

Regular-expression based tokenization

- Remember NLTK's plain-text corpus reader was using a different word tokenizer than nltk.word_tokenize():

```
>>> import nltk
>>> help(nltk.corpus.reader.PlaintextCorpusReader)
Help on class PlaintextCorpusReader in module nltk.corpus.reader.plaintext:

class PlaintextCorpusReader(nltk.corpus.reader.api.CorpusReader)
 |  PlaintextCorpusReader(root, fileids, word_tokenizer=WordPunctTokenizer(pat
tern='\\w+|[^\\w\\s]+', gaps=False, discard_empty=True, flags=re.UNICODE|re.MU
LTILINE|re.DOTALL), sent_tokenizer=<nltk.tokenize.punkt.PunktSentenceTokenizer
object at 0x0000020B5D61A940>, para_block_reader=<function read_blankline_bloc
k at 0x0000020B5D63D9D0>, encoding='utf8')
```

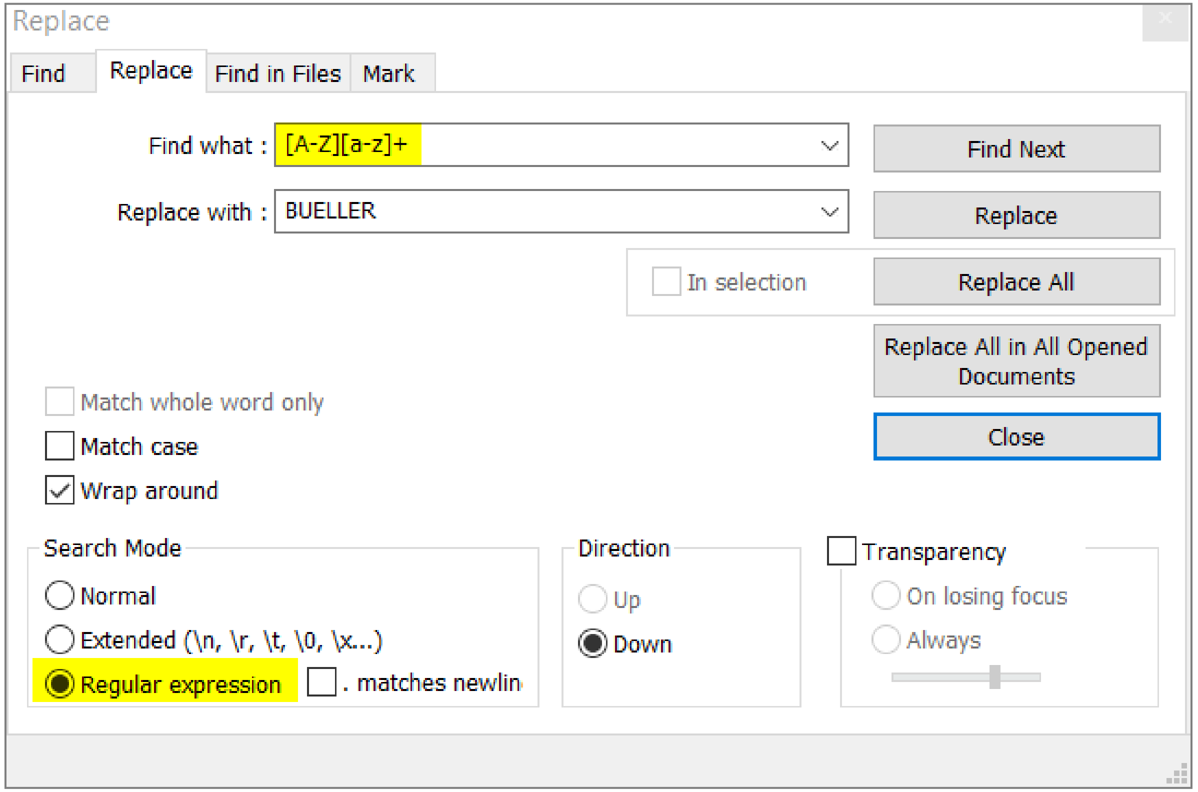
`\w+|[^\w\s]+`

What sort of tokens does this produce?

Regex IRL

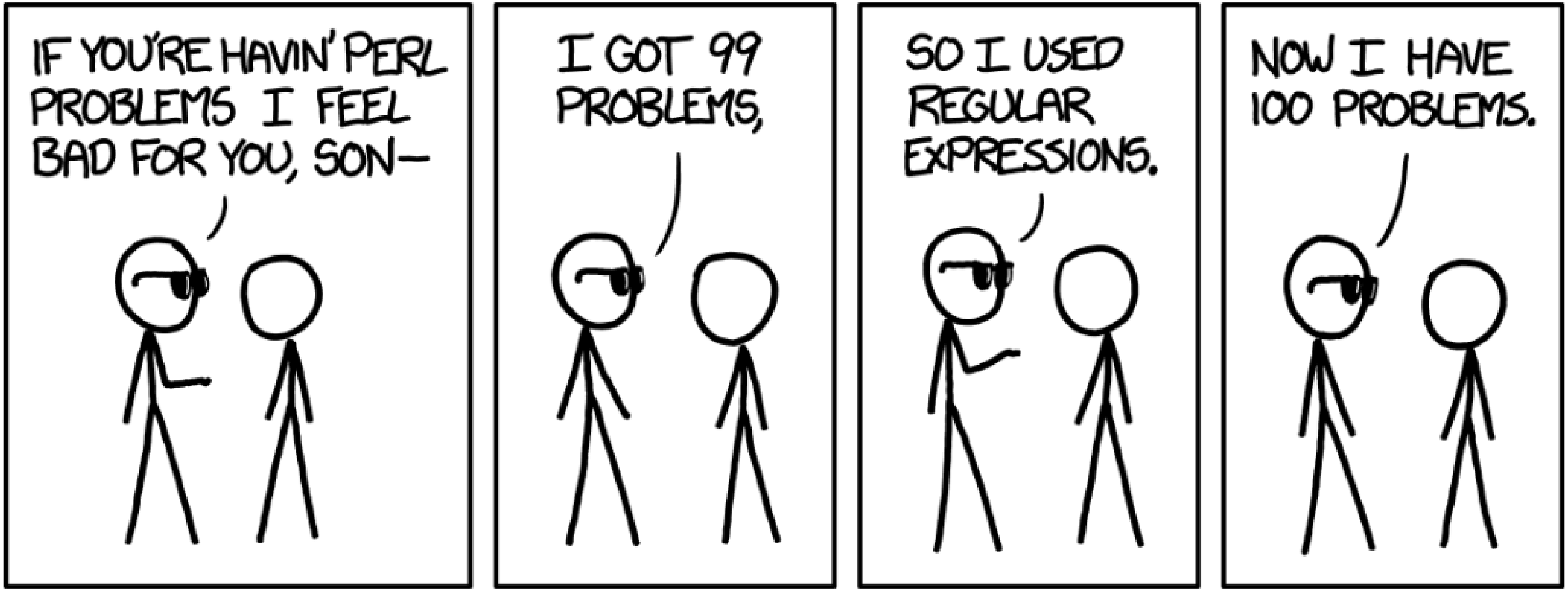
- **Regex-based string substitution** is an extremely common and useful operation.
  - Most text editors provide regex-based search-and-replace capability.

Notepad++ →



99 vs. 100 problems

- https://www.explainxkcd.com/wiki/index.php/1171:_Perl_Problems



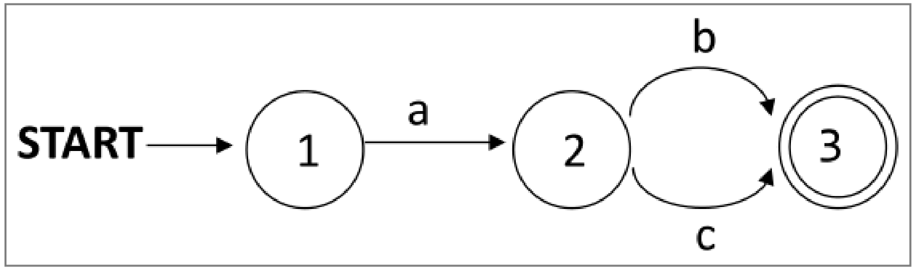
Regular expression pitfalls

- When you were composing regex's for Jobs's Wikipedia entry, you were able to visually confirm what your regex does and does not match.
- In real-life application of regex, **you do not have that luxury**.
  - You do NOT see what your regex failed to match.
    - Hard to debug when you don't even know what's missing (false negatives)!
  - You do get to see positive matches. However:
    - if your search pulls up a huge number of matches, you can't manually go through them to make sure that there are no false positives.
- Regular expressions are very **powerful**, and it takes time and practice to truly master them. Until you have, always be mindful and thoroughly test your regular expressions.

Homework 5: Regex in Python

- Compiling a regular expression through `re.compile()` turns

  `a(b|c)` into



Finite-State Automata (FSA)

- Multi-word proper noun phrase

  ('Steve Jobs', 'Apple I', 'Mac OS', 'The Walt Disney Company'):

  `[A-Z]+[a-z]*( [A-Z]+[a-z]*)+`

Matches 'A', 'AB', 'Abcd', 'ABcd', etc.

A single space is repeated alongside a word

Also: whole regex has to be in ( ) for group capture

Regular expressions vs. automata

- **Regular expression**
  - A compact representation of <u>a *set* of strings</u>

    `/(have|has|had)( n?ever)? been/`

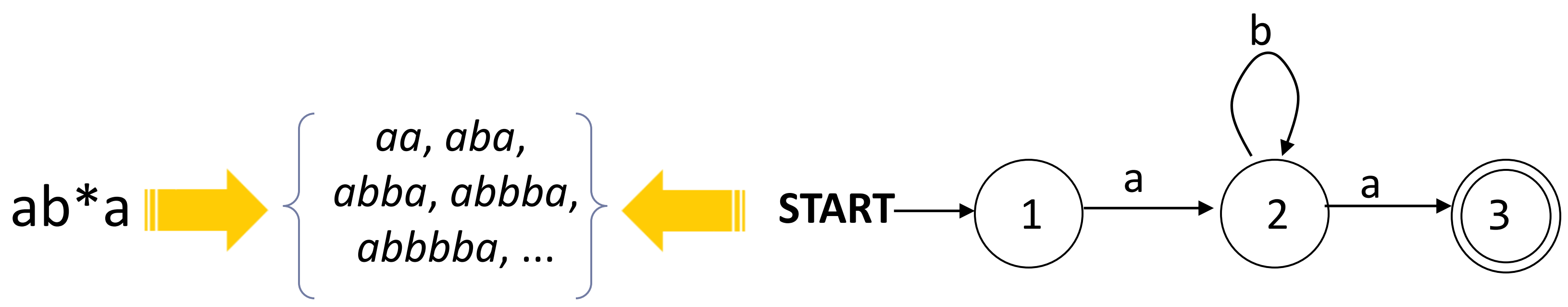
    represents a set of 9 strings.

    `/(have|has|had)( \w+)* been/`

    represents a set of *infinite* number of strings.
- Regular expressions as a formalism have a different incarnation in the form of **finite-state automata**:

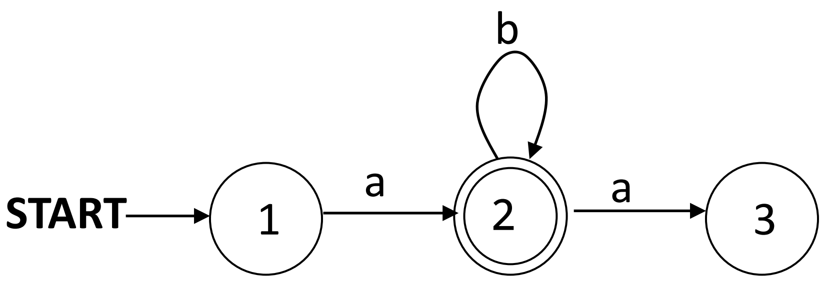
Regular expressions vs. FSA

- Regular expressions and FSA are <u>equivalent</u>.
  - For any regular expression you compose, there is a corresponding FSA.
  - Any FSA can be converted to a corresponding regular expression.
  - How do you define equivalence?
    - A regular expression represents <u>a set of strings</u>.
    - A FSA accepts/generates <u>a set of strings</u>.
    - ← If the two sets are *identical*, the regular expression and the FSA are *equivalent*.

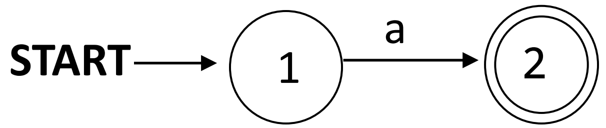


Finite-State Automata

- A **finite-state automaton** (FSA, also called a finite-state machine) is a mathematical model of computation
  - It consists of:
    - **A set of states**. One state is initial; each state is either final (=accepting) or non-final.
    - A set of **transition arcs** between states with a **label**.
    - The machine starts at the **initial state**, and then transits to a next state through an **arc**, reading the **label**
    - When the input string is exhausted, if the machine is at a final state (*ab*), then the string is **accepted/generated**; if not (*aba*), it is **rejected**.
    - Input string is also rejected when it cannot be completely processed. (*b*, *aaa*)



4.12 (*Language and Computers*)



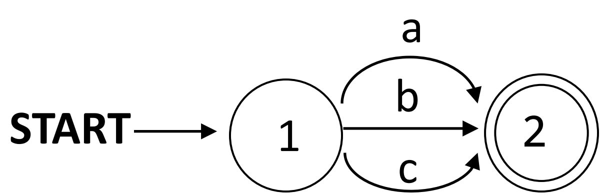
- Accepted by this FSA?

  `''` `'a'` `'aa'` `'b'`

- Equivalent regular expression?

  `/a/`

4.13



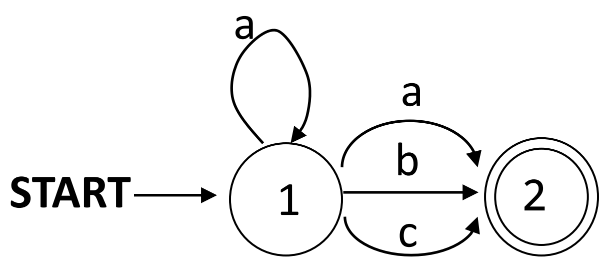
- Accepted by this FSA?

  `''`   `'a'`   `'b'`   `'c'`   `'ab'`   `'bc'`   `'aa'`

- Equivalent regular expression?

  `/a|b|c/`

4.14



- Accepted by this FSA?

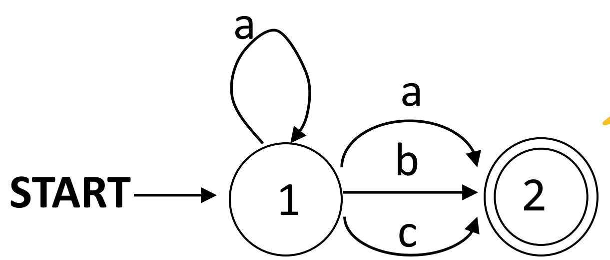
  `''` `'a'` `'aa'` `'aaa'` `'b'` `'bc'` `'abc'` `'ba'`
  `'aba'` `'ab'` `'aac'`

  Also: 'c', 'ac', 'aaac', ...

- Equivalent regular expression?

  `/a*(a|b|c)/`

4.14



Non-deterministic: Multiple choices on reading a single arc label. String is accepted if at least one successful path exists.

A non-deterministic FSA can be algorithmically converted (LINK) into an equivalent deterministic FSA.

- Accepted by this FSA?

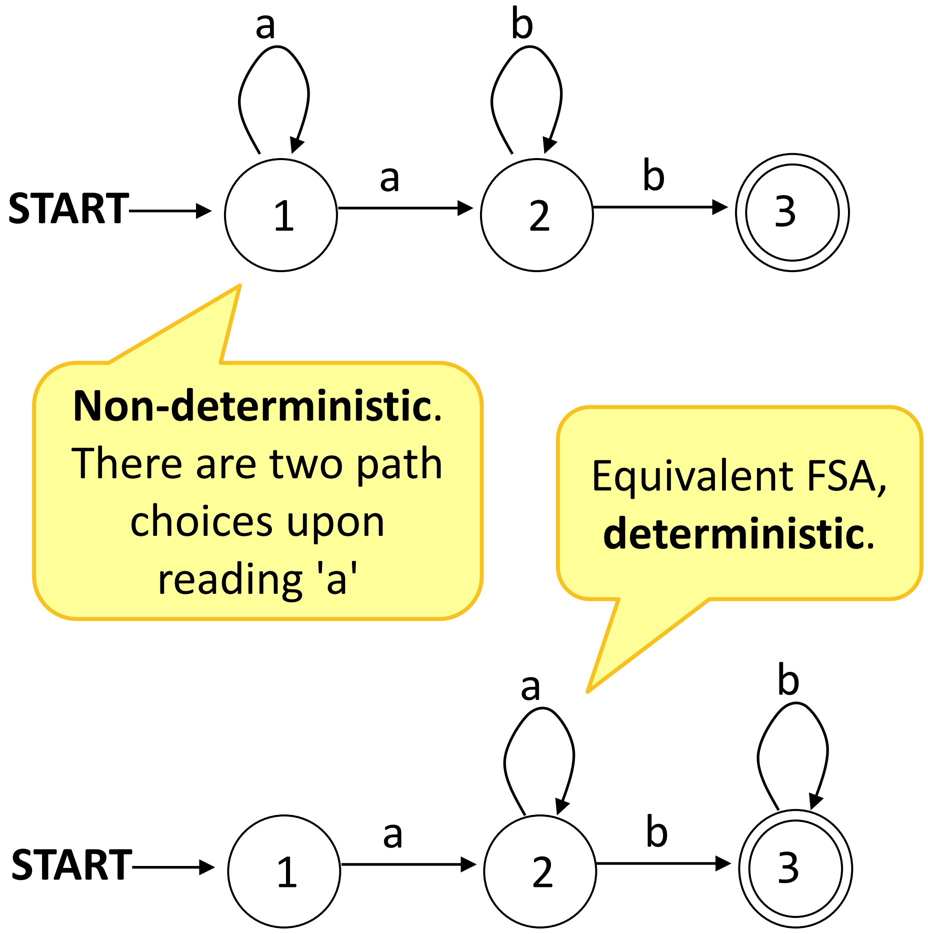
  `''` `'a'` `'aa'` `'aaa'` `'b'` `'bc'` `'abc'` `'ba'`
  `'aba'` `'ab'` `'aac'`

  Also: 'c', 'ac', 'aaac', ...

- Equivalent regular expression?

  `/a*(a|b|c)/`

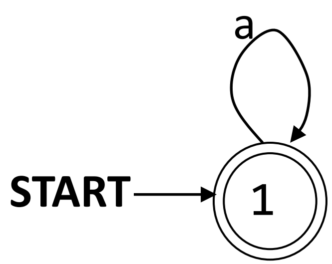
Deterministic vs. non-deterministic FSA



- Which string(s) does this FSA accept?
  - Answer: ab, aab, abb, aaab, aabbb, aaaaabbbbbb, ...
- What is its RE equivalent?
  - Answer: a+b+

A non-deterministic FSA can be algorithmically converted (LINK) into an equivalent deterministic FSA.

4.17



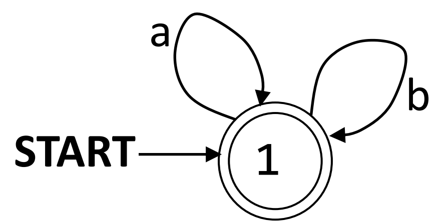
- Accepted by this FSA?

  `''` `'a'` `'aa'` `'aaa'` `'aaaa'` `'b'` `'ab'` `'baaa'`

- Equivalent regular expression?

  `/a*/`

Also: 'aaaaa', 'aaaaaa', ...



- Accepted by this FSA?

`''` `'a'` `'aa'` `'aaa'` `'aaaa'` `'b'` `'ab'` `'baaa'`

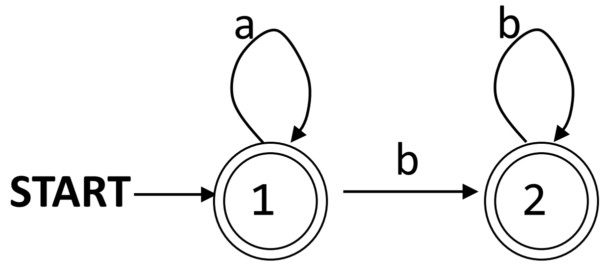
- Equivalent regular expression?

`/(a|b)*/`

If a and b are the only alphabet, equivalent to **`/.*/`**

Basically *any* string made of a and b!

4.18



Wrong FSA shown in the textbook. ← This is the correct FSA that matches the regex.

- Accepted by this FSA?

'' 'a' 'aa' 'aaa' 'b' 'ab' 'aabb' 'baa' 'bab'

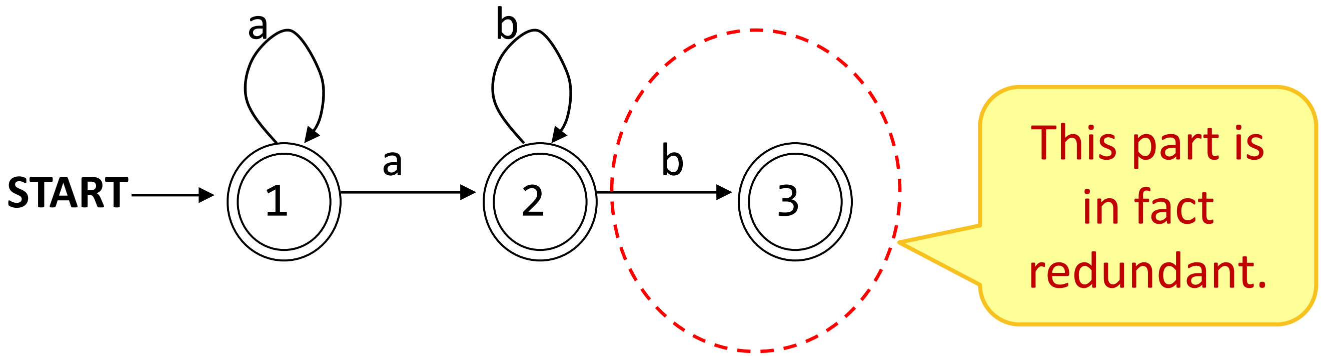
Also: 'bb', 'abb', 'aaabbbbb', ...

- Equivalent regular expression?

/a*b*/

Any number of a's followed by any number of b's

4.18 (as in the textbook p.115)



- Accepted by this FSA?

  '' 'a' 'b' 'ab' 'ba' 'aa' 'aab' 'aaabb'
  'aaaaaa' 'bbbbbb' 'bbbaa' 'aaaab' 'abbbbbb'

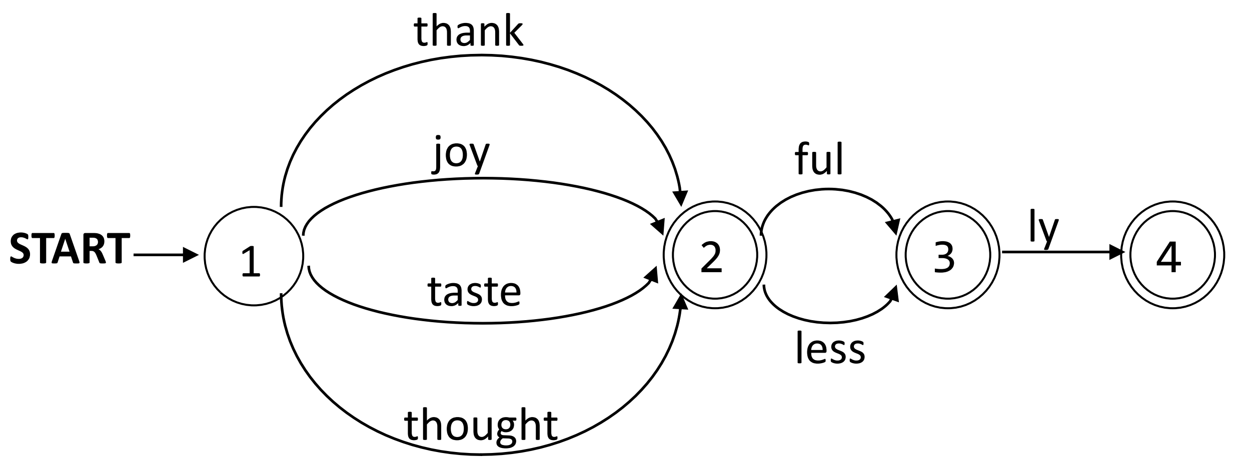
- Equivalent regular expression?

  /a*|a+b+/ /a*(ab*)?/

If b is going to occur at all, it must be preceded by at least one a

English morpho-syntax as FSA

- Arc labels (=vocabulary): *English morphemes*
- Set of accepted strings: *legitimate English words*

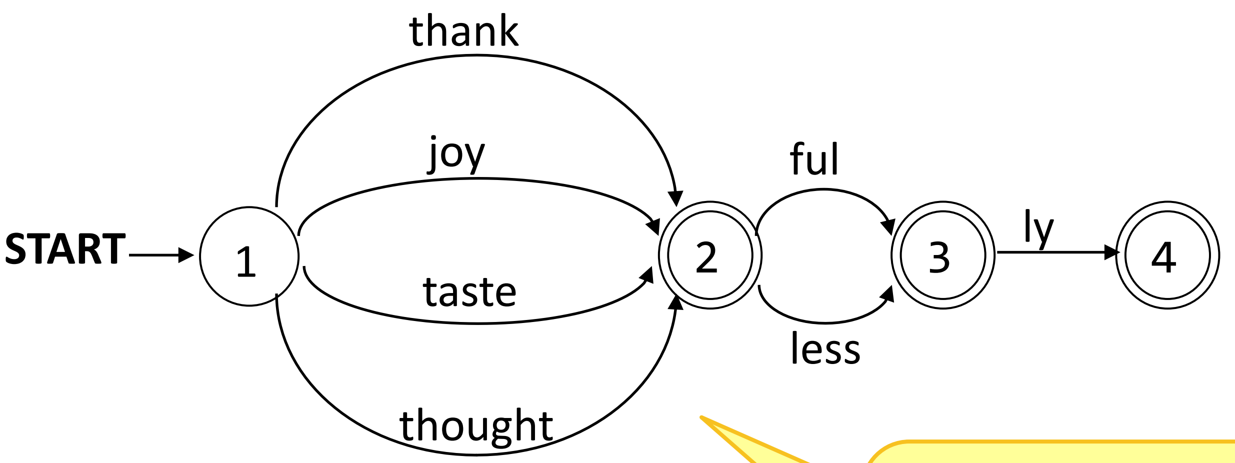


- Which words are in this language?
- Which are not?
- What's the corresponding regular expression?

  `/(thank|joy|taste|thought)((ful|less)(ly)?)?/`

English morpho-syntax as FSA

- Arc labels (=vocabulary): *English morphemes*
- Set of accepted strings: *legitimate English words*



Can we implement the ENTIRE English morphological grammar this way, i.e., as a FSA?

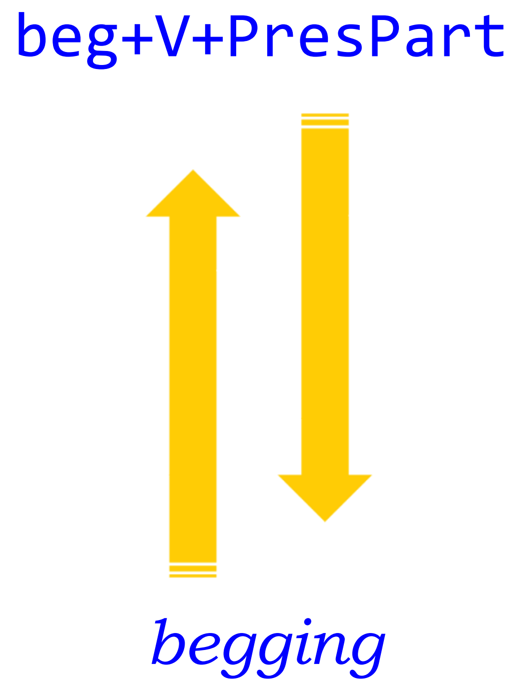
- Which words are in this language?
- Which are not?
- What's the corresponding regular expression?

```
/(thank|joy|taste|thought)((ful|less)(ly)?)?/
```

Computational morphology

- Morphological parsing/analysis
  - Input: a word
  - Output: an analysis of the structure of the word

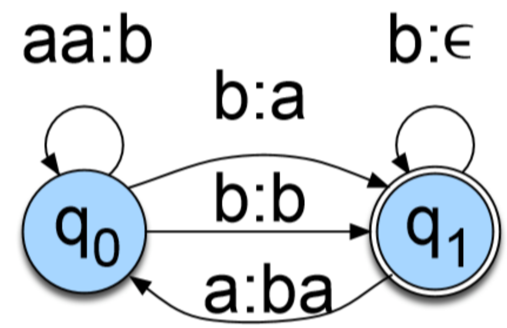
- Morphological generation
  - Input: an analysis of the structure of the word
  - Output: a word



FST: Finite-State *Transducer*

- FSA consists of:
  - **A set of states**. One state is initial; each state is either final (=accepting) or non-final.
  - A set of **transition arcs** between states with a **label**.
  - The machine starts at the **initial state**, and then transits to a next state through an **arc**, reading the **label**

- FST transition arcs have two levels: UPPER and LOWER.
  - FSTs have two sets of finite alphabets for each level.
  - Transitioning involves reading both upper and lower labels.



Upper side and lower side in FST

- Upper side (=underlying form)
- Lower side (=surface form)

Confusing? Yes, but upper/lower comes from the dictionary lookup analogy.
You **look up** "begging" to find it's a present participle form of "beg"

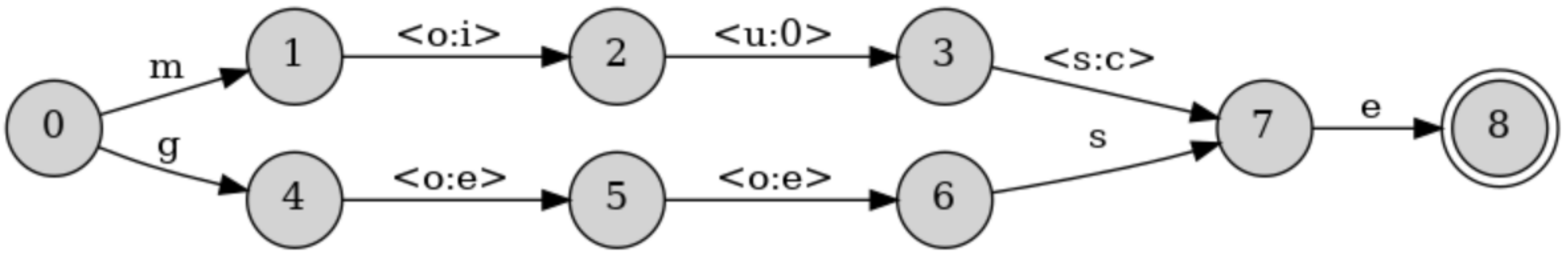
Jurafsky & Martin (ed.2) Ch.3

- Lecture continues, based on the book chapter
- Posted on Canvas. Make sure to review!

3 WORDS & TRANSDUCERS

Introducing: foma

- https://fomafst.github.io/
- A compiler of finite-state machines (FSA and FST)
  - FSA: you already know
  - FST: Finite-State **Transducer**



  - A modern incarnation of Xerox's classic FST suite: XFST and LEXC.

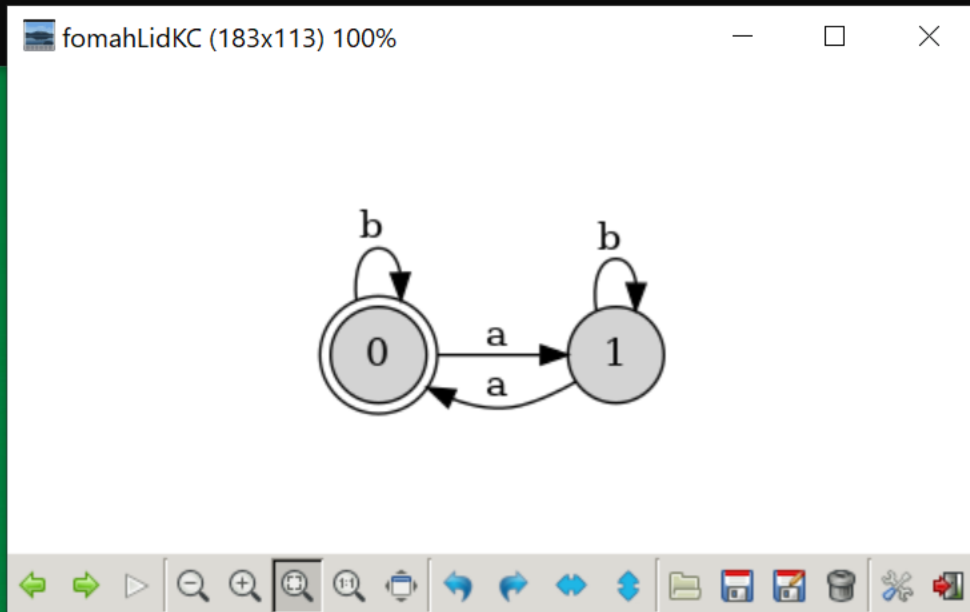
regex in foma: pitfalls

- Foma takes regular expression syntax from Xerox's FST tools, which incorporate many linguistic rule conventions
- Foma's regex syntax differ from the standard (Perl, Python) syntax in some key aspects, most notably:
  - ? → () in foma
  - () → [] in foma
- Additionally, foma adopts multi-character symbols; SPACE is meaningful.
  - "abc" is a single symbol, "a b c" is three symbols concatenated
- Refer to:
  - https://github.com/mhulden/foma/blob/master/foma/docs/simpleintro.md#regex-basics

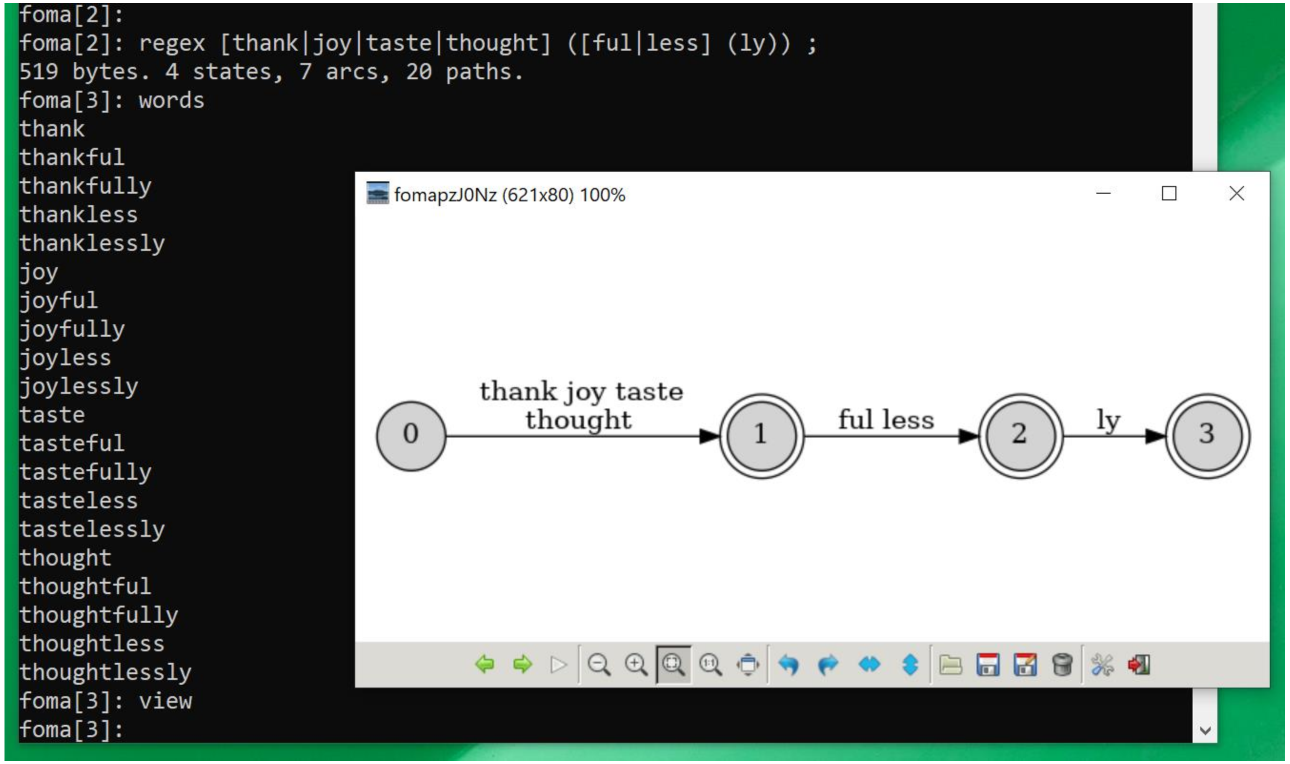
Foma can compile FSA from regex

```
foma[0]:
foma[0]: regex b* [a b* a b*]* ;
303 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: net
Sigma: a b
Size: 2.
Net: 429C4D03
Flags: deterministic pruned minimized epsilon_free
Arity: 1
Sfs0:   b -> fs0, a -> s1.
s1:     b -> s1, a -> fs0.
foma[1]: view
foma[1]:
```

Regular expression is compiled into a deterministic FSA



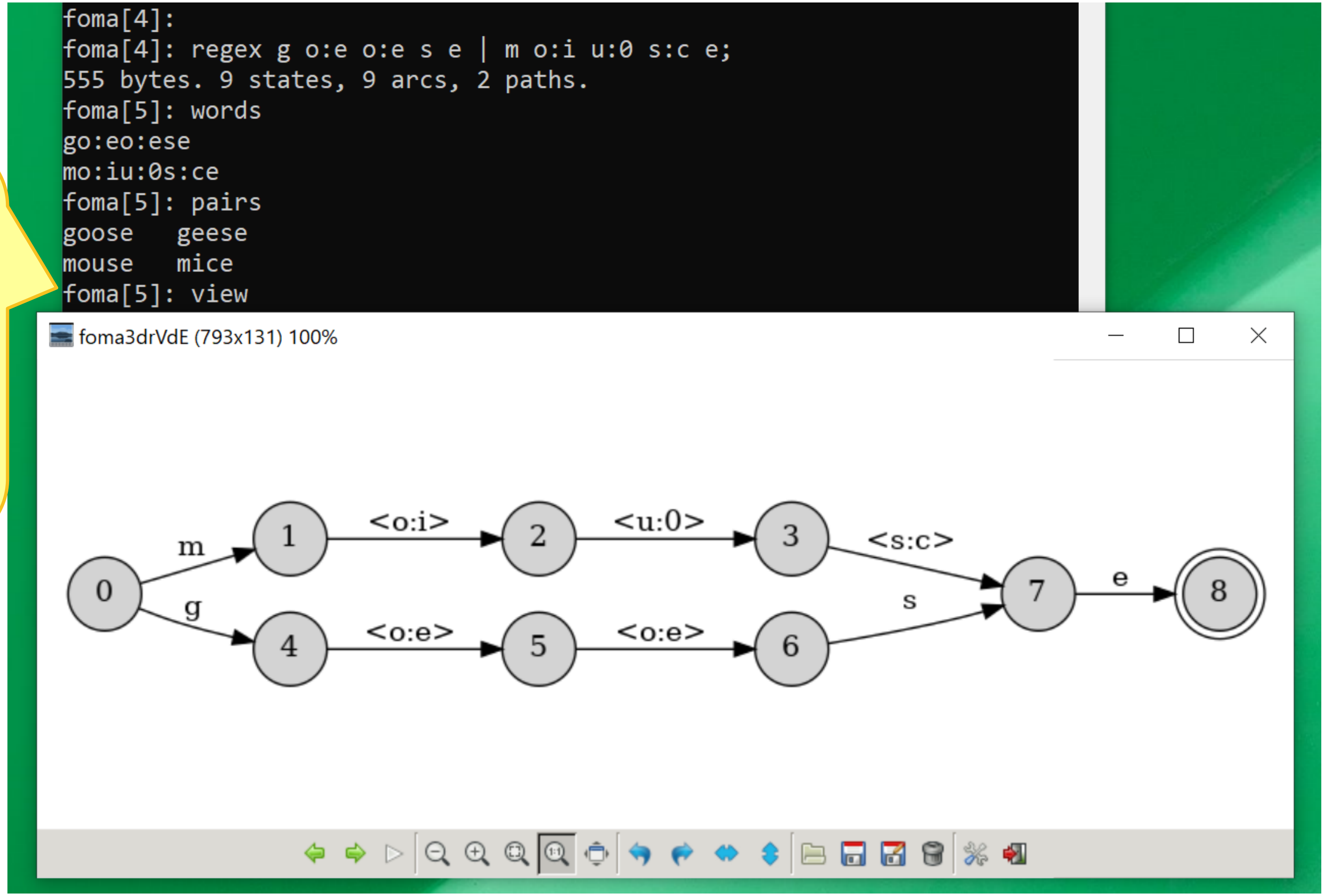
English morpho-syntax as FSA



- Here, "thank", "ful", etc. are construed as multi-character symbols.
- When building a morphological parsers, we don't normally treat morphemes are such. (WHY?)

geese and mice

"view" won't work on Win and OS X. Workaround details in Exercise 8. See Windows workflow, Mac OS workflow.



Wrapping up

- Exercise 8 out
  - Install foma and try it out

- Tomorrow (Wed) 6pm: PyLing! In 2818 CL.

- Thursday: more on Morphology and FST

- What class to take in Spring? → Next slide

Coming soon (hopefully):

Computational Linguistics Certificate

- **Pre-reqs (LING & CS shared):**
  - LING 1578 (phonetics), LING 1777 (syntax), LING 1682 (semantics) *or* LING 1267 (sociolinguistics)
  - COMPINF 401 (intermediate Java), CS 445 (algorithms and data structures 1)
  - STAT 1000 (applied statistics) or equivalent (such as LING 1810)
- **Required content courses:**

LING & CS shared:
LING 1330 Intro to Computational Linguistics CS 1684 Bias and Ethical Implications in AI (or CS 590 for LING majors)

LING majors/minors:*	CS majors/minors:
LING 1340 Data Science for Linguists LING 1810 Stats *or* LING 1269 Variation & Change 1 elective 1 capstone (2-3 credits)	CS 1671 Human Language Technologies CS 1571 Intro to AI *or* CS 1675 Intro to ML 1 elective 1 capstone

* Maximum of 8 credit overlap allowed with LING major/minor